National University of Singapore
School of Computing
CS1010S: Programming Methodology
Semester I, 2018/2019

Recitation 6
Lists, Searching & Sorting

Python

1. *[]* - list constructor
   By itself, creates an empty list. Initialize with elements in this manner:

   ```
   [x1, x2, ..., xN]
   ```

2. *list* - `list(`<*sequence*>`)`
   Takes in a *sequence* type and converts it into a list. If no sequence is provided, i.e. `list()`, an empty list is returned.
3. Common Sequence Operations (works on lists, tuples and strings):
   (a) *len(x)* - Returns the number of elements of list x.
   (b) *element* in *x* - Returns `True` is *element* is in x, and `False` otherwise.
   (c) for *var* in *x* - Will iterate over all the elements of x with variable var.
   (d) *max(x)* - Returns the maximum element in the list x.
   (e) *min(x)* - Returns the minimum element in the list x.
4. Mutation Operations for list *lst*:
   (a) *lst.append(x)* - Modifies list by adding an element x.
   (b) *lst.extend(x)* - Modifies list by adding a another list x.
   (c) *lst.copy()* - Returns a shallow copy of *lst*.
   (d) *lst.reverse()* - Modifies *lst* by reversing it.
   (e) *lst.insert(i, x)* - Inserts element x at index i.
   (f) *lst.pop()* - Removes the last element of *lst* and returns it.
   (g) *lst.pop(i)* - Removes the element at index i in *lst* and returns it.
   (h) *lst.remove(x)* - Modifies list by removing the first occurence of the element x.
   (i) *lst.clear()* - Empties the list *lst*.

Problems

1. Evaluate the following expressions:

```
many_things = [1, 'a', ('I', 'can', 'have', 'tuples', 'in', 'lists')]
print(many_things)

numbers = [2, 3, 4]
print(numbers)

concatenated = many_things + numbers
print(concatenated)

appended = many_things.append(numbers)
print(appended)
print(many_things)

extended = many_things.extend(numbers)
print(extended)
print(many_things)

many_things[0] = 7
print(many_things)

can_be_indexed = concatenated[2]
print(can_be_indexed)

can_be_indexed_multiple_times = concatenated[2][1]
print(can_be_indexed_multiple_times)

a_shallow_copy = concatenated[:]
print(a_shallow_copy)
print(a_shallow_copy == concatenated)
print(a_shallow_copy is concatenated)

woops = a_shallow_copy[2]
print(woops)
print(woops is can_be_indexed)

singleton = ['blah']
print(singleton)
```

2. In lecture, we implemented selection and merge sort, but we did so without using indices. The "typical" way to implement sorts is to use list indices. Implement the following sort:

   (Bubble Sort) Start at index 0, iterate down the list exchanging two pairs of elements if the second is smaller than the first. Rinse and repeat until there are no swaps required during a given pass.

   What is the space and time complexity for your implementation?

3. **(In-place Selection Sort)** Now, implement selection sort, which was described in class, as an in-place sort by using list indices, i.e. write a function `selection_sort` that takes as its argument an unsorted list and modifies it so that it is sorted according to the comparison operator $<$.

   What is the space and time complexity for your implementation?

4. Given the following list:

```
students = [
    ('tiffany', 'A', 15),
    ('jane', 'B', 10),
    ('ben', 'C', 8),
    ('simon', 'A', 21),
    ('john', 'A', 15),
    ('jimmy', 'F', 1),
    ('charles', 'C', 9),
    ('freddy', 'D', 4),
    ('dave', 'B', 12)]
```

   (a) How would you sort the list of students by name in reverse alphabetical order?

   (b) How would you sort the list of students by letter grade, followed by name in alphabetical order?

   (c) How would you obtain a tuple of all the names with fewer than 6 characters?

   (d) How would you obtain a tuple of pairs, where the first element is a letter grade and the second is the number of occurrences of that letter grade?